MASTER IN INNOVATION AND RESEARCH IN INFORMATICS
High Performance Computing
Final Master Thesis

Code generation for the dataflow-based *xsmll* runtime

Author: Daniel Peyrolon Lago
Supervisor: Xavier Martorell Bofill
Director's department: Computer Architecture

FACULTAT D'INFORMÀTICA DE BARCELONA (FIB)
UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC) –
BarcelonaTech October, 2017

Contents

Abstract

This document presents the report of the Master's Thesis Code generation for the dataflow-based *xsmll* runtime developed in the context of the Master in Innovation and Research in Informatics.

In this report we analyze the possibilities that there exist for the OmpSs programming model to be mapped onto the xsmll runtime system. The OmpSs programming model developed at BSC is based on a set of directive-based constructs providing task-based parallelism. The *xsmll* runtime system is being developed at the University of Siena. Both institutions are partners in the AXIOM EU project.

We define and describe the implementation of the code generation phase for the Mercurium compiler, show the limitations we have found both in the input codes that we can transform, and in the capacity that the Mercurium compiler analysis passes currently have. In order to check the correctness of our generated code, we also execute and evaluate some programs compiled with the new phase, making use of the xsmll runtime.

Chapter 1

Introduction

This project has been performed as part of the bigger AXIOM project. The *AXIOM* (acronym for Agile, Extensible I/O Module) project tries to design and manufacture with European funds a single board computer. This board has to be energy efficient, use Open Source software, use a FPGA (Field-Programmable Gate Array) to perform complex calculations, it should integrate support for Arduino "shields" (which are boards using a hardware specification to connect peripheral devices to the Arduino board). Additionally, the boards have to be easily used as a cluster.

With that objectives in mind, some ideas have been explored, and that includes the design and development of a system called *xsmll*, which is a *SDSM* (Software Defined Shared Memory) system which strives to have good performance for clustered environments. Since in order to use *xsmll* there is a need to write code for a specific API, automatic code generation for the API was considered, and the in-house compiler at *BSC* (*Mercurium*) was proposed as the preferred tool. This is the reasoning behind the development of this thesis.

The AXIOM project has a set of so-called *Work Packages*, and each one of them tries to push the project completion forward in a specific direction. The work packages for this project are:

WP1 Management of the project

WP2 Dissemination and exploitation

WP3 Scenario definition and application porting

WP4 Programming model

WP5 Runtime and Operating System

WP6 Architecture implementation

WP7 Evaluation and design space exploration

In the context of the AXIOM project, this work pertains to *WP4* (Programming model). As stated in [8] and [7] this work package deals with:

1. Providing compiler and runtime support in order to program the *AXIOM* cluster with the *OmpSs* programming model, and extend it if necessary. (The OmpSs programming model is described in section 2.1 (OmpSs Description)).

2. The instrumentation design and implementation for the *FPGA* devices in the board.

3. Compiler design exploration to support the distributed cluster environment using *xsmll*. (Which is also described in section 2.2 (*xsmll* description).

This work has been done in the context of the last element in the list.So the general objective is to map the task-based OmpSs Programming Model onto the execution environment defined by the *xsmll* runtime interface. This objective is split into the following sub-objectives:

- Determine if it is possible to convert task-based OmpSs programs to code using correctly the *xsmll* API.

- Determine what shortcomings this transformation has, with respect to the OmpSs application and the compiler analysis available in the Mercurium compiler.

- Implement a set of transformations that will perform the actual code transformations.

- Test and validate the code transformations.

- Evaluate the resulting code with benchmarks.

Chapter 2

Programming Models

This section will describe and provide simple examples of the two programming models that had to be used in the making of this thesis. These are the *OmpSs* programming model, and the programming model implicitly described in the *xsmll* specification.

2.1 OmpSs Description

We will start with the already-known programming model OmpSs, as it is much better known that *xsmll*. We will start describing a little bit the paradigm it uses as well as the ideas used for its development, then proceed to describe how it is usually run, and finally, explain how it is used with a simple example.

2.1.1 Introductory description

OmpSs is a programming model being currently developed at *Barcelona Supercomputing Center* which at the beginning aimed to push the developments made with the *StarSs* (Star SuperScalar), bringing concepts from the superscalar processors into software management of tasks. Currently, OmpSs is being used as a forerunner to develop new extensions to OpenMP, which has already taken benefit for several OmpSs features, like task dependences, priorities, reductions, `tasksloop`, etc.

The OmpSs programming model tries to empower the programmer to transform easily sequential to parallel code through the use of annotations, just like OpenMP. The annotations will describe tasks, and synchronization mechanisms. Each task is the basic unit of work, and they may have dependences, which allows the programmer to describe the data flow of the program, so that –at runtime– the program knows the order in which to execute each task. Although not recommended, barriers (`taskwait`) may be used to assert that the tasks have finished execution before continuing with sequential execution.

Even so, there should be only one barrier (`taskwait`) at the end of the parallel computation.

The biggest difference between OmpSs and OpenMP is *how* parallelism is controlled. The traditional OpenMP model uses *fork-join* parallelism, while OmpSs uses a *thread-pool*. This means that in the former case, a set of threads will be used to execute the code in parallel (fork), and when the parallel execution finishes, only one thread will remain (join). In the latter case though, there will be a set of threads available to the runtime so that they can perform computation in parallel at any moment.

With the traditional OpenMP model the user has to define all the regions of code where he/she wants parallel regions, and define explicitly how this parallelism is handled in the code. The OmpSs programming model instead simplifies this creating an implicit set of threads at the beginning of the program execution, thus liberating the programmer from defining what the parallel regions will be (thread-pool). These threads will then execute all the different tasks asynchronously while respecting all the dependences between tasks. This increases the ease of parallelizing applications, which is considered a hard endeavor.

Given the nature of the programs that are described through the use of tasks and dependences, it makes sense to the developers to push forward the programming model to support heterogeneity in order to get better performance when executing applications. This implies supporting different devices and accelerators, handling development all the way from the programming model (i.e. describing if a task has to be executed on a GPU, or any different device) to the runtime (i.e. actually supporting the data management to comply with the annotations written by the programmer), at the moment, this is currently one of the directions in which OmpSs is being developed.

2.1.2 Implementation

The OmpSs programming model is currently supported thanks to a compiler –Mercurium–, and a runtime system:

Mercurium compiler The Mercurium compiler performs all the needed transformations in order to support the OmpSs programming model, and given a set of tasks, it will generate the code to make that tasks work with the defined runtime.

Runtime systems OmpSs is currently supported by two different runtime systems:

Nanos++ Is the runtime library that creates and manages the thread-pool for the parallel application, and provides all the primitives

needed to support the programming model, like task creation, data movement, data execution... etc.

Nanos6 Is a new runtime being developed at BSC currently, that strives to improve the performance of Nanos++ with a different architecture, and reducing the overhead of the runtime services by redesigning the interface with a focus in simplicity.

2.1.3 OmpSs example

We provide an example of OmpSs implementing a *dot-product* algorithm. This example is taken from the OmpSs exercises at PATC OmpSs courses ([2], [4]).

```
double
dot_product(long n, long chunksize,
            double A[n], double B[n])
{
  long actual_size;
  int j = 0;
  double result = 0.0;

  const long n_chunks = n/chunksize +
                        (n % chunksize != 0);
  double *C = malloc (n_chunks*sizeof(double));

  for (long i=0; i<n; i+=chunksize) {
    actual_size = (n-i >= chunksize)?chunksize:n-i;

    #pragma omp task firstprivate(j, i, actual_size) \
                     in(A[i;actual_size], \
                        B[i;actual_size]) \
                     inout(C[j;1])
    {
        C[j]=0;
        for (long ii=0; ii<actual_size; ii++)
            C[j]+= A[i+ii] * B[i+ii];
    }

    // OmpSs: Accumulate C[j] into result.
    #pragma omp task firstprivate(j) in(C[j;1]) \
                     commutative(result)
    result += C[j];
    j++;
  }

  // OmpSs: Assert task ending before return.
  #pragma omp taskwait

  return(result);
}
```

Listing 1: OmpSs dot product example

The first directive (at line 16 of the example) that we can find at listing 2 (First task of OmpSs example) is:

```
#pragma omp task firstprivate(j, i, actual_size) \
        in(A[i;actual_size],B[i; actual_size]) \
        inout(C[j;1])
```

Listing 2: First task of OmpSs example

This directive defines a task. The associated code is outlined and created as a task. Since the next lines open an explicit scope, the code in this scope is what will be executed with the task.

The next clause is `firstprivate`, this is called a *datasharing* clause. This kind of clause is used to describe how the variables with the corresponding values are going to be used. There are three different datasharing clauses, which are:

private Each thread will have a unique variable whose value is not copied by default (it will not be initialized, and its value will be unspecified).

firstprivate Similar to private, but the value of the variable will be copied into the private thread.

shared All the threads will share its value, and access directly to the variable –usually through a pointer–.

In this case, the variables `j`, `i`, and `actual_size`, are set as firstprivate variables, which means that its value will be copied into the task.

The next clause in the directive is `in(A[i;actual_size])`, this is the first dependence of the task. The dependences are used to explicitly define the order in which tasks have to be executed, by defining how the variables are going to be used. There are three dependence clauses:

in The values of the variables that are `in` dependences will be used, but not overwritten.

out The values of these variables will only be used to write, and not read.

inout The variables will be used both to read and write.

This means that the task is going to use `A[i;actual_size]` just to read its value. We are also seeing for the first time the syntax to define as a dependence a set of positions of a given array. This form is called an extended lvalue at [3]. There are two ways to define them:

array[initial:final] This form describes as a dependency all the elements of an array from the initial to the final one, and both are included in the range.

array[position;size] This one describes as a dependency all the elements of the array from position to (position+(size-1)), and both are included.

Finally, the last clause defines an inout dependency for the task, and uses the extended lvalue form to add a dependency that takes just `C[j;1]`, which is nothing more that `C[j]`.

The next directive in the code (at line 28) is:

```
#pragma omp task firstprivate(j) in(C[j;1]) \
                 commutative(result)
```

Listing 3: Second task of OmpSs example

The only new element in this task is the clause `commutative(result)`. [3] specifies that the tasks that handle the `result` variable will be able to execute in any order –but not in parallel– granted that all sibling tasks that deal with `result` have finished its execution.

Finally, the last directive in the example (at line 35) is:

```
#pragma omp taskwait
```

Listing 4: Third task of OmpSs example

Which is just a barrier to ensure that all tasks have finished before returning from the function.

Given that this document does not try to describe OmpSs in a thorough manner, we refrain from explaining any further feature of OmpSs, specially given that all of its features needed for the thesis have been already described. There are many more OmpSs features that can be used. If the reader wants to learn more about this programming model, we encourage the reader to take a look at [2].

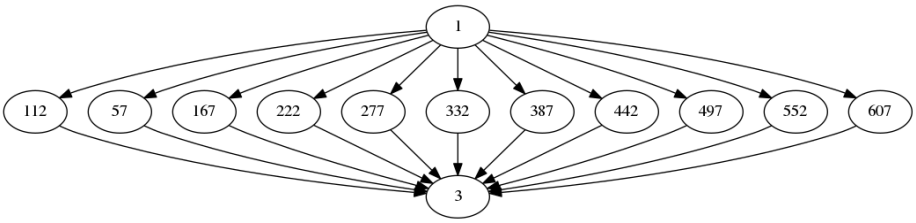

Figure 2.1: Dependency graph for the example

2.1.4 OmpSs dependences

There's one cardinal thing to describe even further. This is how dependences are created and used with OmpSs. Given the next example at listing 5 (Simple tasking example with dependences), there are three tasks. This example performs a parallel summation of two vectors, storing the result of the sun on another different vector.

First task The first task deals with the initialization of the two vectors that will be summed (`A`, and `B`), its `out` dependences are the two initialized vectors.

Second task The second tasks uses as an `in` dependency the `A` and `B` vectors, and `C[i]` as an output dependency. Given that the task is inside a loop, there are `10` different task instances. All of them depend on the first task.

Third task The third task is just an empty dependency that has `C` as an `in` dependency. It therefore depends on all the instances of the second task.

Given all these three tasks, we can build a graph that represents all this tasks (in this case, we're using the Mercurium compiler to build it). It can be seen at figure 2.1 (Dependency graph for the example). This is important and will be taken into account throughout the whole text.

```c
int main() {
    int A[10], B[10], C[10];

#pragma omp task out(A, B, C)
    {
        for (int i= 0; i < 10; i++) {
            A[i] = B[i] = i;
        }
    }

    for (int i = 0; i <= 10; i++) {
#pragma omp task in(i, A[i], B[i]) out(C[i])
        C[i] = A[i] + B[i];
    }

#pragma omp task in(C)
    {
        // Save results or print
        ;
    }


#pragma omp taskwait
}
```

Listing 5: Simple tasking example with dependences

2.2 *xsmll* description

The *xsmll* layer (which stands for *eXtended Shared MEmory (XSM) Low Level specification*) is a runtime system designed with the idea of managing threads, data consistency and synchronization between nodes in a cluster environment. Given that it has been designed for cluster environments, its design is that of a *Software Distributed Shared Memory* (*SDSM*) system. In order to properly understand the reasoning behind its design we need to take a brief look at how SDSM systems work and their performance constraints.

2.2.1 SDSM systems

For starters, we have to define what a SDSM system does. As the name indicates (Software Distributed Shared Memory), its a system that has a shared logical memory space distributed among systems in a cluster environment. This logical memory space can be addressed in any node in the cluster, and the system has to be able to detect that and perform all the necessary data

management in order to fetch the data, or write it in its place in its actual memory location, while keeping a coherency policy for data management (which depends on configuration or system design). This allow its users to design and create applications that work seamlessly in a distributed system without any explicit data management code, which eases development.

As stated in [6], the classic memory management in SDSM systems is performed at a page granularity, using custom page-fault handlers, that will perform the actual data management through the network. While a simple design, it performs poorly, since memory access time is too big. A number of techniques to try to solve this (data prefetch, and a relaxed consistency policy) have been developed through time, but none of them amortizes the memory access time completely.

A well-known problem of SDSM systems is false-sharing, which is a performance-degrading usage pattern that occurs when a system manages and shares data which is partially modified. Since the mechanism to share and manage the data will work at a certain data block size, the *whole block* will be treated like as it were also modified, making the system to copy its whole data block from one system to another –which ideally would happen when the whole data block is modified–. This is also a classical problem in cache design.

This set of unsolved problems have made the research community to stop trying new ideas to overcome this problems, and instead, have focused on systems with explicit synchronization mechanisms (i.e. MPI, OpenMP...), or they have tried to solve (up to now, unsuccessfully) these issues with object-based SDSM systems, where the user has to manage the coherence policy at a per-data-structure level, which makes the users be aware of the coherence policies and underlying architecture (at the expense of making development much harder).

At [6], the authors performed an experiment to verify that page-granularity SDSM systems on modern hardware does not perform ideally. To that end, they ported the *JUMP* system –a SDSM system– to a modern ARM architecture, and tested its scalability (with two boards). Its scalability was negative for two problems out of three, and the authors used that results as a justification to explore different designs for clustered environments (i.e. as *xsmll*).

2.2.2 *xsmll* design

As described at the beginning of this section, *xsmll*, is a runtime that strives to provide thread scheduling and synchronization for distributed systems. Its basic idea is to only start threads when their data is ready to be handled, instead of expecting the programmer to handle the data management part of a distributed system –using synchronization code such as mutexes–, and data

management code. This model is an extension of the Dataflow Threads, which was developed for the Teraflux project ([5]), and is also further explained at [6].

The *xsmll* runtime represents any given thread with the *xthread* construct. This is the basic computation block, and represents a thread being executed in any node of a given *xsmll* cluster. All xthreads are represented through the `xtid_t` type –this is called an xthread descriptor from now on–. An xthread is a function that has no parameters, and returns no value:

```
typedef void (*xthread_t)(void)
```

Listing 6: Type of xthread

Given that an xthread is a function that does not accept or return values it must fetch data in a different way. That is handled by the input and output frames. Both are a buffer in local memory that contains the data for the xthread, or succeeding xthreads. The xthread needs to have all the data in its frame before starting.

Every xthread has an associated *synchronization count*, which is an integer that should be initialized to the number of inputs of the xthread, and whenever an input is written, is should be decreased through the use of a function. When finally decreased to zero, the runtime executes the xthread corresponding to the synchronization counter.

We can see at table 2.1 (*xsmll* Specification functions (as seen in [6])) a definition of the whole API of the *xsmll* runtime. As we can see, this simple API does not allow us the same ease when dealing with parallelism as OmpSs. The idea of this project is to support automatic transformation from OmpSs code to code leveraging the *xsmll* API.

Since the API description is not enough to understand how the runtime works, we are going to explain an example similarly to OmpSs –by taking an example and proceed to describe how it uses the API–. But we first need to explain a relatively complex part of the runtime that is not properly explained by the API definition, its shared memory.

2.2.3 *xsmll* dependences

Provided that we will need to implement dependent tasks, we will take advantage of the xthread dependence counter to implement the task graphs. Also the final *taskwait* can be implemented with the same support.

Function	Syntax and explanation
xschedule	`xtid_t xschedulez(xthread_t ip, uint32_t sc, uint32_t sz)` Schedules the xthread whose name is `ip`, sets its synchronization count to `sc`, and allocate a frame of `sz` bytes.
xdestroy	`void xdestroy()` Called at the end of the xthread to deallocate its resources.
xsubscribe	`void* xsubscribe(xtid_t tloc, uint32_t regionstart, uint32_t regionsize, uint8_t type)` Subscribes a memory region that is going to be shared by xthreads with some r/w protection specified by `type`. The region of memory that is going to be subscribed is defined by an offset (`regionstart`), and its size (`regionsize`), both expressed in bytes, and aligned to 8 bytes. It returns the pointer to the region, and is stored in `tloc`, which is an element of the frame of a given xthread). When a region of memory is "subscribed", it means that an element of its frame will be changed to a pointer to the shared memory region that we want to use.
xpublish	`void* xpublish(void* regptr)` Forces the update of the memory region pointed by `regptr`, which has to be previously subscribed.
xpreload	`void* xpreload()` Returns a pointer to the input frame of the xthread. It's used at the beginning of the xthread.
xpoststor	`void* xpoststor(xtid_t tid)` Returns a pointer to the frame of the xthread `tid`. Used from out of the xthread in order to pass values to it.
xdecrease	`void xdecrease(xtid_t tid, int n)` Used to decrease the synchronization counter of the xthread `tid`.
xtmbegin	`uint64_t xtmbegin(uint64_t s1, uint64_t s2)` Begins a transaction on the transactional memory region pointed by `s1` and having a length `s2`. Returns 1 if the transaction can start, and 0 otherwise. (Unused in the making of this project).
xtmend	`uint64_t xtmend(uint64_t s1, uint64_t s2)` Ends a transaction on the transactional memory region pointed by `s1` with a length of `s2`. (Unused in the making of this project).

Table 2.1: *xsmll* Specification functions (as seen in [6])

This is implemented by creating xthreads with a non-zero synchronization counter, passing the xthread descriptor to another xthread through its frame, and when an xthread finishes, using the xdecrease API to decrement the counter of the dependent xthreads.

Implementing this technique automatically in our compiler requires a way to detect the dependences between tasks. In order to do so, we have leveraged a pre-existing phase of the Mercurium compiler, the so called analysis phase, as will be seen in chapter 7 (*xsmll* lowering phase).

2.2.4 *xsmll* shared memory

The API provides the programmer with the tools to use shared memory between threads. This is done through the use of the *xsubscribe* and *xpublish* functions. Since we are developing a compiler phase we do not really care about the actual implementation of these functions, but we do care about using them correctly. With that in mind, let us delve further in what these functions do.

`xpublish()`

The `xpublish()` function prototype is:

```
void* xpublish(void* regptr);
```

This first function (xpublish), as stated by the table 2.1 forces the update of the memory region pointed by `regptr`. Basically, the runtime will take care of updating the value of that memory region in the shared memory and copy its new value from the xthread as necessary (to the other nodes in the system, if it must). This is used in order to ensure that xsubscribed values will be updated (otherwise, the shared memory will be unchanged and work performed by the xthread will be in vain).

`xsubscribe()`

The other function is `xsubscribe()`, its function prototype is:

```
void* xsubscribe(xtid_t tloc, uint32_t regionstart, uint32_t
↪  regionsize, uint8_t type);
```

This function has 4 parameters:

tloc This first parameter is the xthread whose memory we want to subscribe. While its type is `xtid_t`, it will not point at an xthread. It will be a pointer to the position of the frame of the xthread (the `xtid_t` value returned from `xschedulez` points to the beginning of the xthread frame).

As we will see in the example, this first parameter is the result of calling a macro that will add the xthread ID to the offset of an element in its frame. Therefore, its real value is a direct pointer to the element of the frame whose memory we want to subscribe.

regionstart This is the offset of the shared memory where we want to start to subscribe the data. Its starting position.

regionsize The size of the region that we want to subscribe.

type This represents the xthread permissions of the thread when dealing with the memory. It has 3 possible values: read permissions, write permissions and read and write permissions.

When using the function we are telling the runtime that we want to share this memory region with other xthreads by using a pointer, reserving `regionsize` bytes in the shared memory, and changing the pointer value in the frame to point to this region of memory. We can then work with that region of memory, and at the end of the xthread, we should call `xpublish` to publish that changes in the memory. That will basically update the memory region across all xthreads.

2.2.5 *xsmll* example

We provide a simple of example of an xthreads application in order to properly describe how the application using *xsmll* will work at listing 7 (*xsmll* matrix multiply example).

```
1  #define N 200
2  #define BLOCKSZ 25
3  #define DATA uint64_t
4  #define SIZE(M,N) (N*M*sizeof(DATA))
5
6  #define RB_SZ SIZE(N,N)
7  #define RA_SZ SIZE(N,BLOCKSZ)
8  #define RC_SZ SIZE(N,BLOCKSZ)
9
10 #define RC_OFF(j) (            SIZE(N,j))
11 #define RB_OFF(j) (  SIZE(N,N)+SIZE(N,j))
12 #define RA_OFF(j) (2*SIZE(N,N)+SIZE(N,j))
13
14 typedef struct {xtid_t xr;} owm_matrix_mul_s;
15 void owm_matrix_mul() /* Compute C=A*B.  */
16 {
17   const owm_matrix_mul_s *fp = xpreload();
18   xtid_t xr = fp->xr;
19
```

```
20    for (int j=0; j<N; j+=BLOCKSZ) {
21      // Schedule the actual multiplying function.
22      xtid_t bm = xschedulez(bmmul, 4, sizeof(bmmul_s));
23      xsubscribe(XDST(bm, bmmul, A), RA_OFF(j), RA_SZ,
        ↪  _OWM_MODE_R);
24      xsubscribe(XDST(bm, bmmul, B), RB_OFF(0), RB_SZ,
        ↪  _OWM_MODE_R);
25      xsubscribe(XDST(bm, bmmul, C), RC_OFF(j), RC_SZ,
        ↪  _OWM_MODE_W);
26
27      bmmul_s* bm_fp = xpoststor(bm);
28      bm_fp->xreport = xr;
29      xdecrease(bm,4);
30    }
31    xdestroy();
32  }
33
34  typedef struct {
35    DATA *A; DATA *B; DATA *C;
36    xtid_t xreport;} bmmul_s;
37  void bmmul()
38  {
39    const bmmul_s *cfp = xpreload();
40    const DATA *A = cfp->A;
41    const DATA *B = cfp->B;
42    DATA *C = cfp->C;
43    int i, j, k;
44
45    for (j = 0; j < BLOCKSZ; j++) {
46      for (i = 0; i < N; i++) {
47        DATA t = 0;
48        for (k = 0; k < N; k++) {
49          t += A[j * N + k] * B[k * N + i];
50        }
51        C[i + (j * N)] = t;
52      }
53    }
54    xpublish(C);
55    xdecrease(cfp->xreport,1);
56    xdestroy();
57  }
```

Listing 7: *xsmll* matrix multiply example

This example has been taken from the matrix multiply example provided at

the *xsmll* Git repository. We have taken just two functions since it is enough to provide a good explanation of how to work with *xsmll*.

Let us start describing what `owm_matrix_mul` does. It firsts gets a pointer to its own frame through the use of `xpreload()`, and the type of the pointer is that of an `owm_matrix_mul_s`, that is a struct defined just before the function. This struct contains a `xtid_t` object, which is the xthread descriptor of the report function (this function deals with checking if the program has been correctly executed, and print a message accordingly, –its code has been omitted–). The idea of having this field in the struct is to pass it to the `bmmul` function so that it calls `xdecrease()` when it has finished. When it has been decrease `N/BLOCKSZ`, the synchronization count will be zero, and its execution will be triggered.

Following in the code, we have the loop that will be executed `N/BLOCKSZ` times. This loop has a call to `xschedule()` to schedule a call to bmmul. It has as a first parameter the function to be scheduled, as a second parameter the initial value of the synchronization counter (`4`), and lastly, the size of its frame (which is `sizeof(bmmul_s)`, where `bmmul_s` is the struct defined just before the `bmmul` function). After the call to xschedulez, there is a call to `xsubscribe()`, its first parameter is a call to the macro `XDST()`. This is defined at the *xsmll* header as:

```
#define XDST(_xtid,_fun,_off) \
        ((_xtid)+(uint64_t)(offsetof(_fun ## _s, _off)))
```

Which resolves in the first `xsubscribe()` call to:

```
bm + (uint64_t) offsetof(bmmul_s, A)
```

This macro uses three parameters, but only one the first references a runtime object. The other two are just the struct definition, and the name of its field. This macro is important, since we need to generate code with this behavior, but the Mercurium compiler is not able to generate macros, as it will be seen in section 4.2 (Macro generation constraint).

The macro calculated the actual position of the A pointer in the frame of `bmmul`. This pointer will be set with the value of the location of the A array in the shared memory of *xsmll*, which is written previously in a different function. In order to point to these arrays we use a combination of the `RA_OFF()` and `RA_SZ()` macros, which are defined at the beginning of the function. With `RA_OFF` we point to the offset of the location in shared memory, and with `RA_SZ` we declare the size of the array, as a last parameter to `xsubscribe()`, we have `_OWM_MODE_R`, which defines the permissions of the xthread to use the data (read, write, or both)

After the calls to xsubscribe, we call `xpoststor()` to get the frame of the `bm` xthread, and set the `xreport` field with the value of the `xr` value (the field that was taken through `xpreload()`), and call `xdecrease()` to start the xthread. Please note that the value `4` is totally arbitrary (the authors could have used `1` as well). This mechanism is the one described at section 2.2.3 (*xsmll* dependences). Finally `xdestroy()` is called to deallocate the resources used by this specific xthread.

For the xthread `bmmul` there is not much to note, besides that it uses the value passed at the end of the `owm_matrix_mul` xthread, and calls `xdecrease()` on that specific xthread (which has been previously scheduled). Another interesting point is the fact that it calls `xpublish(C)` to publish the result of the function. That is because the xthread whose counter is decreased has been subscribed to that array and we need to call `xpublish()` in order to update the value of the array.

2.2.6 *xsmll* usage

In order to use the *xsmll* runtime, the AXIOM project partners are provided provides the developers with a VirtualBox image ready to be used. This VirtualBox image contains a Ubuntu 14.04.5 LTS distribution that contains all that is needed in order to run it. For now this is what it includes (besides the Ubuntu system):

SimNow simulator This is a simulator written by AMD in order to create a PC platform emulation environment. It is used to run

COTSon This is a tool build on top of the SimNow simulator, written in Ruby from HP-Labs. Its main contribution is to be able to simulate a specific microarchitecture on demand and perform reliable simulation. More about it can be read at [1].

Xsmll runtime and library This is the actual runtime implementation, it is a library and the headers needed to use the *xsmll* API in code. Its examples come with specific `Makefile` used to compile them and execute them with cotson.

dsetools This is a in-house-developed set of shell scripts used to manage experiments running with *xsmll*. The user can specify the inputs of the programs and will take care of changing the `Makefile` when needed in order to properly run the examples. In the end we opted for not using it and rely on modifying the Makefiles by hand.

When you execute a correct *xsmll* program this is what happens under the hood:

- Once you have a proper program to execute, its code should be in the `test` folder, with the rest of the examples provided of the *xsmll* runtime.

- The Makefile should be modified to set the `TESTS` variable with the correct filename, the wanted number of nodes that should be used, and size of the shared memory (using the `OWMSZ` makefile variable), and the inputs should be specified by setting the `SZ` variable.
- Once all that is set, when running `make run`, make will save a script at `/tmp`, compile the program including the *xsmll* headers, and compile it with its library, with the `--wrap,main` flag among others –this makes the `main()` function in the program to be wrapped with another function specified at the library, which will actually start the runtime–, and execute cotson using the generated script. This script will take care of copying the binary into the virtual machine and running it.
- When it has been executed, cotson will take care of printing runtime information, and `makefile` will print a digest of it.

Chapter 3

Methodology

This section explains the methodology for the development of this project. It was developed using the waterfall model.

Before designing the phase for the Mercurium compiler, we first tried to understand how the *xsmll* runtime actually worked by reviewing the examples given with the runtime. There were two valid examples, one which implemented a *dot-product* algorithm, and another one that implemented a *matrix multiply* algorithm.

Once the model was understood, –we not only read and execute the examples, but also wrote some additional applications, and tried to determine how to implement the final synchronization at the end of the parallel region (`taskwait`)–, we also had to properly start defining the transformations needed in order to generate *xsmll* code from OmpSs code. The design of these transformations is described in section 5 (Code generation).

When we had a clear idea of how these transformation had to be performed, we then proceeded with a phase design and implementation in order to perform the transformations. This design was constantly improved and tinkered with, since for every test that we wrote, we potentially had to change some details of the implementation. This is also described at section 7 (*xsmll* lowering phase)

During this developmental process, we wrote tests that we used to verify the results of the development, and we refined further our design when the design failed until we fixed most of its shortcomings and bugs –just as the waterfall model implies–.

Chapter 4

Limitations

In the development of this project we realized that not every OmpSs application could be ported directly to *xsmll* without changes. There are a variety of reasons for such limitations, that come from the limitations in the *xsmll* API, and inherent limitations in the Mercurium compiler that we need to take into account.

We explain all the changes that should be done to an original OmpSs application so that it can be correctly compiled into OmpSs code in this chapter.

4.1 *xsmll* API limitations

There are some limitations in the expressiveness of the *xsmll* API that will hinder the options available to generate code.

Lack of barrier primitives

One limitation we found out very soon in the design phase is that since we want to support the most dataflow-like parts of OmpSs in order to generate *xsmll* code, we could not support barrier constructs, nor OmpSs features that could break a dataflow-like paradigm –like the `final` clause, that uses a condition in order to decide whether to compute a task using a thread of the runtime pool, or compute it sequentially–.

Another limitation that stems from this is that since we can't use barriers, we need to somehow use the dataflow paradigm to start a xthread after a loop. The way this is done in the examples given to us with the runtime is to know in advance how many iterations of the loop will be executed, and use that as the initial value of the synchronization count of the xthread after the loop. Whenever an iteration ends, it will have to use the `xdecrease()` function call to decrement the synchronization counter of the successor xthread. That way, when the last innermost xthread has finished its execution, the successor xthread will be triggered. This behavior is seen at listing 7 (*xsmll* matrix multiply example) with the `report()` function

The other problem associated with this is that we cannot retrieve values generated by an xthread from sequential code. In order to do so, we would need to support a barrier construct, or a way to fetch data from the main thread, and *xsmll* does not provide us with that functionality. Some tricks were tried in order to get over this constraint, but their result was unsatisfactory.

In order to overcome the lack of a direct barrier primitive from *xsmll*, we tried using a "barrier thread". This was a dummy xthread that would act as a barrier. We would set its initial *synchronization counter* to 1 –recall that the xthread would start its execution when the synchronization counter gets to 0–. Its synchronization counter is increased by one every time a previous xthread is triggered, and it is decreased when this xthread finishes. In order to increase the synchronization counter, we added a `xincrease()` function, which increases the synchronization counter for an xthread by a specified amount. Finally, at the original barrier location, the synchronization counter would be decreased by 1 so that its execution would be triggered after the completion of each and every previous xthread.

Since our barrier primitive did not work –due to problems with the runtime–, we had to impose as a constraint that all the values generated by xthreads must be used inside a task.

Inter-iteration dependences

Given the nature of *xsmll* thread descriptors `xtid_t`, we cannot (trivially) use xthreads depending on different iterations of a given loop. A simple example of how this is done with OmpSs is shown in the next snippet of code.

```
for (i = 0; i < n; i++)
{
#pragma omp task in(A[i-1], i) inout(A[i]) shared(A)
        A[i] = A[i-(i == n)?0:1] + calc_offset(A);
}
```

Listing 8: Code with dependences between different iterations

For that reason, when dealing with dependences we flatten the task instance space, making it all have the same dependences, and the same dependant tasks.

One way that this could be supported is by having array of xthread descriptors (one for each iteration), and passing each element of the array to its corresponding xthread instance.

Usage of *xsmll* shared memory

According to the *xsmll* API, we can only use shared memory from *inside* an xthread. Therefore, we can *only* translate code whose shared variables are modified only from a task.

4.2 Mercurium limitations

These are the limitations of the Mercurium compiler that made the project harder.

Lack of interprocedural analysis

The Nanos++ runtime system (which is the runtime that supports OmpSs) receives the data dependences hints as expressed by the programmer, and it computes the dependences between tasks internally. This way, it does not matter if the next task created is on a different application scope.

On the other hand, with *xsmll*, the programmers must express directly the dependences between the tasks. When generating the code for a task, the compiler must know which will be the dependent task to be executed afterwards. Observe that this is not possible without interprocedural analysis, as the compiler will not know which is the next task.

The Mercurium compiler currently lacks the features to perform interprocedural analysis (that was a never an objective of the compiler, since it does not need to perform it in order to support the OmpSs runtime). That forces us to set as a constraint that the transformations must be done for only the tasks in one

function.

Macro generation constraint

The next limitation is due to the Mercurium compiler lacking support for code generation with macros. This is because `gcc` is used to preprocess the code before the Mercurium compiler parses the code, this performs macro substitution and eliminates all the macros, thus lifting some load on the Mercurium compiler. This complicates code generation, as it will be seen.

Loop analysis constraint

As seen in the API limitation, we have to perform loop analysis from the compiler point of view, there are some instances of code in which this won't be possible (i.e. loop limits depend on user input).

Since the phase will be used in controlled environments, we can perform such analysis by ourselves (or by leveraging third-party libraries for the Mercurium compiler), and set as a limitation that the number of iterations of loops containing tasks must be constant.

Task dependency analysis

Since we have to provide the successor tasks at code generation time, the compiler has to perform the dependence analysis between the tasks expressed by the programmer. We are using the analysis passes already present in the Mercurium compiler, partially developed in the PSocrates EU project [9].

4.3 Summary

So, as a summary, these are the limitations we found out when developing this project:

1. The results of the computations must be used in another task.
2. We cannot translate properly any code that uses barriers (taskwait), unless the order of the task is well-defined through dependences.
3. Different *tasked* iterations of a loop must always depend on the same tasks, and they cannot depend on different iterations.
4. All initialization of shared variables must be in a task.
5. All tasks must be in a single function.
6. We cannot generate code that uses macros.

7. If we want to generate correct code that leverages parallelism, we need to perform loop analysis, and use applications that use loops that depend on constants.

8. We must perform some kind of task dependency analysis before generating code.

Chapter 5

Code generation

In this chapter we'll outline the transformations needed to proceed with the translation from OmpSs to the *xsmll* API. First, let's pinpoint some obvious design objectives for the code generation phase:

- For each task, we'll have to generate the `struct` that represents the frame, and its corresponding xthread function.
- We need to generate the `xschedule` and `xdecrease` calls for every xthread.
- We also need to get all the variables that the xthread needs before it starts its execution.
- If there are dependences between xthreads, we *must* be able to generate code that will respect them –otherwise we're just generating code that won't be executed in the specified order–.

We'll demonstrate the points of this chapter with actual code generated by the Mercurium compiler.

5.1 Dependences between xthreads

First of all we need to explain the synchronization mechanisms at our disposal in order to maintain an ordered execution of the tasks in the system –If dependences are not respected, we will never get correct results from scientific applications–. We need to expose this first since it permeates the code generation at every layer.

Given a set of dependences between tasks, we say that a *successor* task depends on a *predecessor* task if the latter needs to execute and finish before the former. With the *xsmll* model, we need to pass the descriptor of the successor xthread to the predecessor xthread –so that it can trigger the execution of

the successor xthread–. This is achieved by adding a field to the frame of the predecessor with that purpose, and calling the `xdecrease()` function it xthread.

We can clearly see that a task will depend on another one if one of its `in` or `out` dependences is an `out` dependency of another previous task. Since our code cannot have barriers (taskwait), as seen in section 4.1 (Lack of barrier primitives), all the dependences will have to be expressed in this way. In order to use them, we use a class that performs the task analysis for us in the Mercurium compiler (as explained in section 4.2 –Task dependency analysis–).

This makes compulsory the scheduling of the successor xthread *before* the predecessor xthread gets executed, which means that we have to generate first the `xschedule()` calls for each task that is going to be executed in that scope. That way, if we need to use the xthread descriptors, we'll have them available.

There is also another issue to take into account. It is the fact that if there is a loop which contains a task, we have to know the number of iterations of the loop because we will have to pass the xthread descriptor of the next task to every instance of the task in the loop. Then, we will call `xdecrease()` from every iteration of the loop, decreasing the synchronization counter of the next task. This implies that we set the number of iterations as the value for the initial synchronization counter of the *next* xthread.

5.2 xthread and frame generation

Since we're dealing with xthreads, for every task, we'll have to define a function which won't accept any parameter and won't return any value, and have the task code embedded in that function. Likewise, we'll have to define a data structure (the one used as a frame for the task). Both the struct and the xthread function must be defined at the global scope of the current compilation unit.

5.2.1 xthread struct

Generating the structure is a relatively simple task. We just need to add all the variables used by the task to a struct. This information is obtained from the programmer, and it uses the dependency and datasharing information. Depending on the datasharing and dependency, we might add elements differently to the `struct`.

in, out, inout All of these variables are automatically added as `firsprivate` variables, unless a datasharing is specified –this is the standard behavior– The behavior for each datasharing is also described in this list.

private All these variables doesn't have to be added to the data structure. In order to maintain the semantics of OmpSs, we just add a variable to the frame stack. This is done simply by adding a declaration of a variable with the same name and type of the variable inside the xthread function.

firsprivate We have to add a field to the data structure sharing the name and type of the firstprivate variable of the task. This field will be set after the `xschedule()` function call using the `xpoststor()` function and an assignment to the field of the xthread –`xpoststor()` returns a pointer to the frame of the xthread–.

shared Similarly to `firsprivate` variables, we have to add a field, which will use the same name as the variable, but instead of sharing its type, it will just be a pointer to that type. This variable will then be used as a memory region inside the shared memory of *xsmll*.

Dependent xthreads We need to pass the xthread descriptors of the tasks that depend on the one for which we are generating the `struct`. This will be further explained through the section, but for now, suffice to say that we will add a variable of the type `xtid_t` for each of these dependent xthreads, and that they are treated as `firstprivate` variables.

5.2.2 xthread function

Besides the creation of the `struct` for the xthread, we also need to generate the function for that xthread. This function, as explained many times, doesn't accept any parameters, and it doesn't return any value. The generation of the function code is divided in three parts, its preface, body, and its epilogue.

Function preface

Initially, we get the xthread's frame (represented by the `struct` whose generation has been already explained), this is done through the use of the `xpreload()` function.

The result of that call is a pointer to the xthread's frame, and the next step for the preface of the function consists in generating definitions of the local variables used in the function, whose values will be taken from the frame. These variables will share the type and *name* of the variables, so that they can be used without any problem in the *function body*.

Lastly, we declare all the private variables of the task that is represented with this xthread. By doing so, the compiler will use the activation record to allocate the memory needed for the variables, whose values will be undefined, as it is defined in the OmpSs specification.

Function body

The next part of the function is its body. This part will have the task's code without just one change: since all the variables that are `shared` are now a pointer to the type of the variable, we have to rewrite the code that uses this variables and make it use a pointer instead, dereferencing the variable each time its used.

Function epilogue

After the preface, and the execution of the task code, we still need to generate code for a few reasons.

- First of all, we need to ensure that the shared variables are correctly propagated through the runtime to the shared memory so that the xthreads that are going to use this variable can use its new value –this is achieved by a call to the `xpublish()` function–.
- The next step is to call `xdecrease()` for all the xthreads that depend on the execution of the current one, the xthread descriptor will be taken from its frame.
- As the last step, we will add a call to `xdestroy()` in order to deallocate the current resources allocated for the xthread.

5.2.3 Function and struct example

Follows a simple example of an xthread function and structure, that contains all of the elements exposed so far. The code uses comments in order to name and show them.

5.3 xthread schedule and start

If we want to schedule and start the xthread we are going to need to generate calls to the `xschedulez()` and `xdecrease()` function calls, –at least one pair for each xthread that we want to execute–. In this section we explain how code is generated for both functions.

5.3.1 xschedule code generation

At listing 10 (xschedule prototype), the reader can see the prototype of `xschedulez()`. Its first argument is an xthread, which (as seen in 2.2.2), is just a function pointer of a function that accepts no parameters, and returns nothing. Its second argument is the original value of the synchronization counter. When set to 0, the execution of the scheduled xthread will be triggered. The final argument is the size of the frame. Usually, we just make a call

```
struct xsmll_struct_1_s  //
{                        // Struct required for the xthread
  xtid_t f2;             //
  int *a;                //
};                       //
struct xsmll_struct_1_s; // Struct declaration

static void xsmll_task_1(void) // Function definition
{
                             // Epilogue
  int b;                     // Private variables
  struct xsmll_struct_1_s    //
    *f1_data = xpreload();   // Getting the frame
  xtid_t f2 = (*f1_data).f2; // Initialization of local variables
  int *a = (*f1_data).a;     //

  *a = 10;           // Body of the xthread
                     // a is dereferenced, since it is now a
                     // shared variable

                     // Epilogue
  xpublish(a);       // Write new value to shared memory
  xdecrease(f2, 1); // Decrement the counter of dependant xthread
  xdestroy();        // Deallocate resources
}
```

Listing 9: Example of code generation for struct and function

to the `sizeof()` operator with the name of the `struct` that represent the frame.

In the design phase of the *xsmll* API, the synchronization counter was intended to represent the number of inputs of the xthread frame. When the fields of the frame were set to the desired value, the user would call `xdecrease()` to lower the value. We instead use the synchronization counter to represent the number of xthreads that need to be executed before its xthread starts its execution. In order to do so, we opted to use the number of xthread on which the current xthread depends, if there are any, or 1 otherwise (how the dependences information is obtained is described at section 5.1 –Dependences between xthreads–).

In order to avoid problems, we opted to prepend all the required `xschedulez()` calls at the beginning of the scope of the task. Doing this, we will ensure to have all xthread descriptors available to pass to any other xthread that may need them, in order to call `xdecrease()` on them. How this is achieved is explained at section 5.4 (Passing variables to an xthread).

```
xtid_t xschedulez(xthread_t ip, uint32_t sc, uint32_t sz);
```

Listing 10: xschedule prototype

5.3.2 xdecrease code generation

At listing 11 (xdecrease prototype), the prototype of `xdecrease()` can be read. Its first argument is a xthread descriptor, and the second argument is an `int` that indicates by how much the xthread synchronization counter must be decreased.

```
void xdecrease(xtid_t xtid, int n);
```

Listing 11: xdecrease prototype

The position where the `xdecrease()` function calls have to be inserted is not as trivial as it seems, since given a successor and predecessor task, we need to schedule the successor before calling `xdecrease()` for the father, because we need to pass the xthread descriptor to the father so that its synchronization counter is decremented. We also need to take into account that all the variables have been passed to the frame of the xthread. In the end, we just opted to append it at the location of the last task of the scope of the task.

Another thing to consider is by how much we decrement the synchronization counter each time we generate a call to `xdecrease()`. Since the synchronization counter represents the number of dependences, we have to generate a call with a value of 1 if it's called from a xthread, and the same applies to the root threads (those which do not depend on any other threads), since they use a synchronization counter with a value of 1 by default, which makes the code generation for xdecrease calls very straightforward.

5.4 Passing variables to an xthread

Given that we want the xthreads to compute something useful, we need to fill the frame of the xthreads. In this section we explain how this is accomplished for `firsprivate` and `shared` variables. We omit `private` variables since we do not pass any value with that datasharing, as their value must be undefined –they are allocated in the activation record of the function, as seen in section 5.2.2 (xthread function)–.

All of the code described in this section must be located at the position where the task was, since it may need some variables that are only available at its location. Furthermore, as explained previously, all the calls to `xschedulez()` and `xdecrease()` need to be done at the start and end of the current scope, respectively, which lets us use pass the value of all the xthread descriptors that we may need to use – they are passed as `firsprivate` variables–.

5.4.1 Firstprivate variables

To pass `firsprivate` variables, we first need to generate a call to the `xpoststor()` function (it uses the `xtid_t` obtained with `xschedulez()`), and returns a pointer to the frame that said xthread is going to use. We can see the prototype of this function at listing 12. We only require to pass an xthread descriptor to it, and it will return a `void *`, which will point to the frame of the xthread. We will assign its return value to e pointer to the `struct` used as its frame.

Once that we have generated a call to such function, we can also generate an assignment to the fields of the `struct` that correspond with `firstprivate` variables.

We can see at listing 13 one example of code generated to pass a `firsprivate` variable.

```
void* xpoststor(xtid_t xtid)
```

Listing 12: `xpoststor()` prototype

```
// Call to xpoststor() in order to get a pointer to the frame
struct xsmll_struct_1_s *f1_frame = xpoststor(f1);
// Actual assignment of a field of said frame
(*f1_frame).f2 = f2;
```

Listing 13: Code generated to pass `firsprivate` variables

5.4.2 Shared variables

In order to use effectively shared variables, we need to set them up in the *xsmll shared memory*. This is done with a call to `xsubscribe()`, and the variables have to be update with a call to `xpublish()` to ensure its propagation at the end of the xthread. The call to this function is added at the function of the

xthread (as seen in 5.2.2).

We'll take one snippet from listing 7 as an example. Please remember that this code comes from an example given to us, and has been written manually, this helps to explain the nuances of the generation of `xsubscribe()` call.

```
xsubscribe(XDST(bm, bmmul, A), RA_OFF(j), RA_SZ, _OWM_MODE_R);
```

Listing 14: Call to xsubscribe() in hand-written code

As we can see at listing 14 –Call to xsubscribe() in hand-written code–, every argument of the function is computed with a macro, but as we saw in section 4.2 (Macro generation constraint), we cannot generate code using macros. Instead, we opted to add the functionality of these macros with inline functions at the *xsmll* header files. We describe how we substituted the macros with functions that implement the same functionality, and show the code of such functions at listing 16 (Functions added to *xsmll* header to substitute macros).

The first parameter corresponds to a call to the `XDST()` macro. Since we can't generate calls to it, we added a `xdst()` function that calculates the address of a specific field in the frame of an xthread. Contrary to the `XDST()` macro (as seen in section 2.2.5 –*xsmll* example –), we need to calculate the address at runtime –not at compile time– so we need an instance of the frame that is used by the xthread. Regrettably, this makes compulsory the generation of a previous call to the `xpoststor()` function in order to get a frame instance –even if there are no `firstprivate` variables–. This way we can calculate its address without the use of the `offsetof()` macro.

As for the second and third arguments of the `xsubscribe` function, the second argument is the offset of the region of shared memory that is going to be used, and the third one, the size of such region. This is calculated at compile time using the size of the field that uses a `shared` datasharing –how it is done will be explained at section 7.4 (`SharedMemAllocator` and `SharedMemVar` classes)–. The results of the calculations are later aligned to 8 bytes through the use of a function called `aligned_size()` –we explained that memory alignment is compulsory for the *xsmll* at section 2.2.2 (*xsmll* design)–.

The last parameter is the macro substitution for `_OWM_MODE_R` (there are two more, `_OWM_MODE_W`, and `_OWM_MODE_RW`). Luckily, all of them only return a value that represents the permissions of the xthread for this shared memory region –which is encoded just by an `int` variable–. We wrote one `owm_mode` function for each macro that returns its value. This allows us to just generate a simple function call.

Lastly, we show one example of how one field of a frame may be set up to use *xsmll* shared memory with code generated by out phase at listing 15 –Code generated to use `shared` variables–.

```
// Call to xpoststor() in order to get a pointer to the frame
struct xsmll_struct_1_s *f1_frame = xpoststor(f1);

// The actual call to xsubscribe. It requires to use
// the result of xpoststor to calculate the address
// of the field that must point to shared memory
xsubscribe(xdst(f1, f1_frame, &(*f1_frame).a), aligned_size(0),
↪  aligned_size(4L), owm_mode_w());
```

Listing 15: Code generated to use `shared` variables

5.5 Working example of code generation

In this section, we will see a complete example of the code generation process to create valid *xsmll* code, using the code generation primitives described previously. At listing 17 –OmpSs example of dependent tasks– we have a simple OmpSs code that uses dependences –required if we want to demonstrate all the code generation features–.

The code only has two tasks, and the second task is a *successor* of the first one. This means that we will have pass the xthread descriptor of the second task to the first one –using a `firstprivate` datasharing–, so that it can call `xdecrease()` from the predecessor function.

We need to generate both a struct and a function for each task. The first task will have a frame with a xthread descriptor, and an `int*` that will point to shared memory. The frame of the second task will have an `int*` that will use shared memory to get the value of `a`, and another field to capture the value of `b` (of type `int`). The `struct`s that have been generated for both tasks can be seen at listing 18.

As for the functions of these two tasks, let us explain how they should be. Its code can be found at listing 19.

First task The code generated for the first task must be as follows:

prologue The prologue of the first task only needs to call `xpreload()` in order to get its frame, and take the values for those variables into local variables. These will be `f2` (the name of the second thread descriptor), and `a` –`int*`, because it is a shared variable–.

```
_INL static xtid_t xdst(xtid_t tid, void* structure, void*
↪  element)
{
/* XDST(_xtid,_fun,_off) ((_xtid)+(uint64_t)(offsetof( _fun ##
↪  _s, _off))) */
     return (xtid_t)(void*)((char*)tid + ((char*)element -
      ↪  (char*)structure));
}

_INL static int owm_mode_w() {
    return _OWM_MODE_W;
}

_INL static int owm_mode_r() {
    return _OWM_MODE_R;
}

_INL static int owm_mode_rw() {
    return _OWM_MODE_RW;
}

_INL static size_t aligned_size(size_t s) {
    return (s+7) & ~0x07;
}
```

Listing 16: Functions added to *xsmll* header to substitute macros

body The body of the first task must be the same code for the task `a = 10;`, except that `a` must now just be a pointer to `int`, because it is shared, so the assignment must be `*a = 10;` instead.

epilogue Finally, since the first task has a shared variable, it must call `xpublish(a);` to propagate its value to the shared memory. After that, since the execution of the first task has ended, it must call `xdecrease(f2, 1);` to trigger the execution of the second task. Finally, it must deallocate its own resources, calling `xdestroy()`.

Second task The code for the second xthread will be very similar, except that now it gets `b` as a `firsprivate` parameter, and it has a type `int`. `a` is still a shared variable and its used as an `int*`, and it doesn't receive any other xthread descriptors. Since there are no changes to store for this task (`a` has an `in` dependency, so we don't have a need to generate a `xpublish()` call for this variable.

The last point left to demonstrate is the code generated to instantiate a task,

```
1  int main()
2  {
3    int a;
4    int b = 10;
5  #pragma omp task inout(a) shared(a)
6    a = 10;
7  #pragma omp task in(a) inout(b) shared(a) firsprivate(b)
8    a = b * 2;
9
10 #pragma omp taskwait
11   return 0;
12 }
```

Listing 17: OmpSs example of dependent tasks

```
struct  xsmll_struct_1_s
{
  xtid_t f2;
  int *a;
};
struct  xsmll_struct_2_s
{
  int b;
  int *a;
};
```

Listing 18: Frames generated for the example

set the values we pass it, and trigger its execution. This can bee seen at listing 20 –20–.

- First of all, the `xschedulez()` and `xdecrease()` calls (the latter only for the first task) are added at the beginning and the location of the last task in main function, respectively.
- The normal code before task execution is the declaration of the variable `a` and the definition of `b`. The first one is not required in the last code generation step of the Mercurium compiler and is not generated. The second one is.
- For the first task, we have a shared variable (`a`), and a `firsprivate` (the xthread descriptor) variables, the first one is set, and the second one is captured and propagated to the frame (by using `xpoststor()`, and `xsubscribe()`).

```
static void xsmll_task_1(void)
{
  struct xsmll_struct_1_s *f1_data = xpreload();
  xtid_t f2 = (*f1_data).f2;
  int *a = (*f1_data).a;
  {
    *a = 10;
  }
  xpublish(a);
  xdecrease(f2, 1);
  xdestroy();
}

static void xsmll_task_2(void)
{
  struct xsmll_struct_2_s *f2_data = xpreload();
  int b = (*f2_data).b;
  int *a = (*f2_data).a;
  {
    *a = b * 2;
  }
  xdestroy();
}
```

Listing 19: Xthreads generated for the example

- For the second task, we have `a` as a shared variable, and `b` as a `firsprivate` variable. The first one is again set through a call to `xsubscribe()` and the second one with an assignment of the frame's field. Since this is the last task of the scope, the required `xdecrease()` calls for the root tasks are added.
- Finally, the rest of the code is left untouched. The taskwait before the end is deleted, as it doesn't translate to any *xsmll* constructs, and it is only required for specific implementation details –which will be explained at chapter 7, *xsmll* lowering phase–.

This is a thorough demonstration of the code generation process for a simple case. In the next section we will explain how this Mercurium compiler phase works internally.

```c
int main()
{
  // Task instantiation.
  xtid_t f2 = xschedulez(xsmll_task_2, 1, sizeof(struct
  ↪  xsmll_struct_2_s));
  xtid_t f1 = xschedulez(xsmll_task_1, 1, sizeof(struct
  ↪  xsmll_struct_1_s));

  // Setting up b
  int b = 10;

  // First xthread frame setting process
  struct xsmll_struct_1_s *f1_frame = xpoststor(f1);
  xsubscribe(xdst(f1, f1_frame, &(*f1_frame).a), aligned_size(0),
  ↪  aligned_size(4L), owm_mode_w());
  (*f1_frame).f2 = f2;

  // Second xthread frame setting process
  struct xsmll_struct_2_s *f2_frame = xpoststor(f2);
  xsubscribe(xdst(f2, f2_frame, &(*f2_frame).a), aligned_size(0),
  ↪  aligned_size(4L), owm_mode_r());
  (*f2_frame).b = b;

  // Trigger execution of f1
  xdecrease(f1, 1);

  // Return
  return 0;
}
```

Listing 20: *xsmll* code generated at `main()` with OmpSs example

Chapter 6

Mercurium runtime architecture

This chapter explains how the runtime of the Mercurium compiler works, and how phases and visitors –the bread and butter for code generation and transformations in the compiler– can be written and implemented.

Obviously, in order to develop a Mercurium phase, the author of a phase also has to modify the build system. While interesting, this is omitted as it isn't relevant for the project deliverable. Also out of the scope of the deliverable has been considered a great part of the internal API of the Mercurium compiler, given that we can provide a good description of how it works without the nitty-gritty details.

6.1 The Mercurium compiler driver

First of all, we need to explain how the Mercurium compiler works internally at a high level. After the `main()` function starts execution, the steps that the Mercurium compiler does are, *a grosso modo*:

- Initializes internal configuration following the arguments and configuration of the program. Also configures and sets some UNIX environmental resources (signals, setting `atexit()` function...).
- Parses all the flags passed to the source-to-source compiler and sets internal models accordingly.
- Calls `compile_every_translation_unit()` function, where for every translation unit:
  - Loads all the phases required by the compiler's configuration (these phases are loaded as external shared objects).

  - Calls the native compiler set in the configuration to preprocess every file.
  - Parses the translation unit (it represents a source code file).
  - Initialize the *DTO* –it stands for Data Transfer Object–, this object implements the homonymous pattern, which is nothing more than a common object that contains information which is shared and used through all the different phases.
  - Runs the *pre-execution* function for all the loaded phases. This function can be set up by the phase programmer so that it does some special initialization for the phase.
  - Checks that the AST that we have so far is correct. This is done before the most important step for this process.
  - Run all the *run*, and *phase_ cleanup* functions of every phase. For instance, when compiling code with OpenMP or OmpSs annotations it will produce code that uses the specified runtime (depending on the switches passed to the Mercurium compiler), to execute the code in parallel.
  - Once all the phases have performed its required work. It calls the native compiler in order to generate actual binaries.

- Once all of this has been done, the next step is to embed all the required object files. This is done in order to put all the object files together after compilation, before linking.
- The next step is to link the required objects. This is done with the help of an external linker.
- Finally, `main()` finishes and the execution of the Mercurium compiler ends.

6.2 Phase structure

Each phase in the Mercurium compiler is represented by a parent class called `CompilerPhase`, this class contains the following the following public functions:

Constructor The constructor of the class is called when the phase is loaded, after having read the configuration files.

Destructor The destructor is called just before finishing the compiler.

pre_run This function is executed before the parsing and typechecking of a translation unit.

run Entry point of the phase with parsed and typechecked translation unit.

phase_cleanup This function is executed to perform all the cleanup after the phase has been executed.

phase_cleanup_end_of_pipeline Similarly to `phase_cleanup`, this function is called to do any extra cleanup, but will be executed at the end of the pipeline, when all the other phases' `run()` function has been executed for a single file.

register_parameter Registers a parameter for the phase. These parameters are used to set variables for the phase.

get_parameters Returns all the parameters of the phase.

setters and getters Used to set and get the phase name and the phase status.

This is the entry point for every phase writer in the Mercurium compiler, and every phase must implement this methods. Obviously, we need to make the heavy lifting of the phase in the `run()` function (possibly some work may be done in the `pre_run()` function as well).

6.3 Nodecl tree

All the code in a translation unit is represented by an enriched tree. This tree is composed of `Nodecl` class instances –this will represent the syntax structure of the translation unit–. The tree is also enriched with symbolic, type, scope, and constant values information, necessary in order to use it to perform code transformations. This will tree will be traversed using Nodecl visitors.

For visualization purposes, we are adding a piece of code and its resultant AST tree representation as given by the Mercurium compiler. The code is at listing 21 (Simple example for AST), and its internal representation is at figure 6.1 (Resultant AST for the example).

```
1 int main() {
2     int a = 2;
3     printf("Hello, world: %d\n", a);
4     return 0;
5 }
```

Listing 21: Simple example for AST

6.4 Nodecl visitors

Now that we have explained how to set up a Mercurium phase, and how it is used from the top level, we also need to expose how to write a visitor

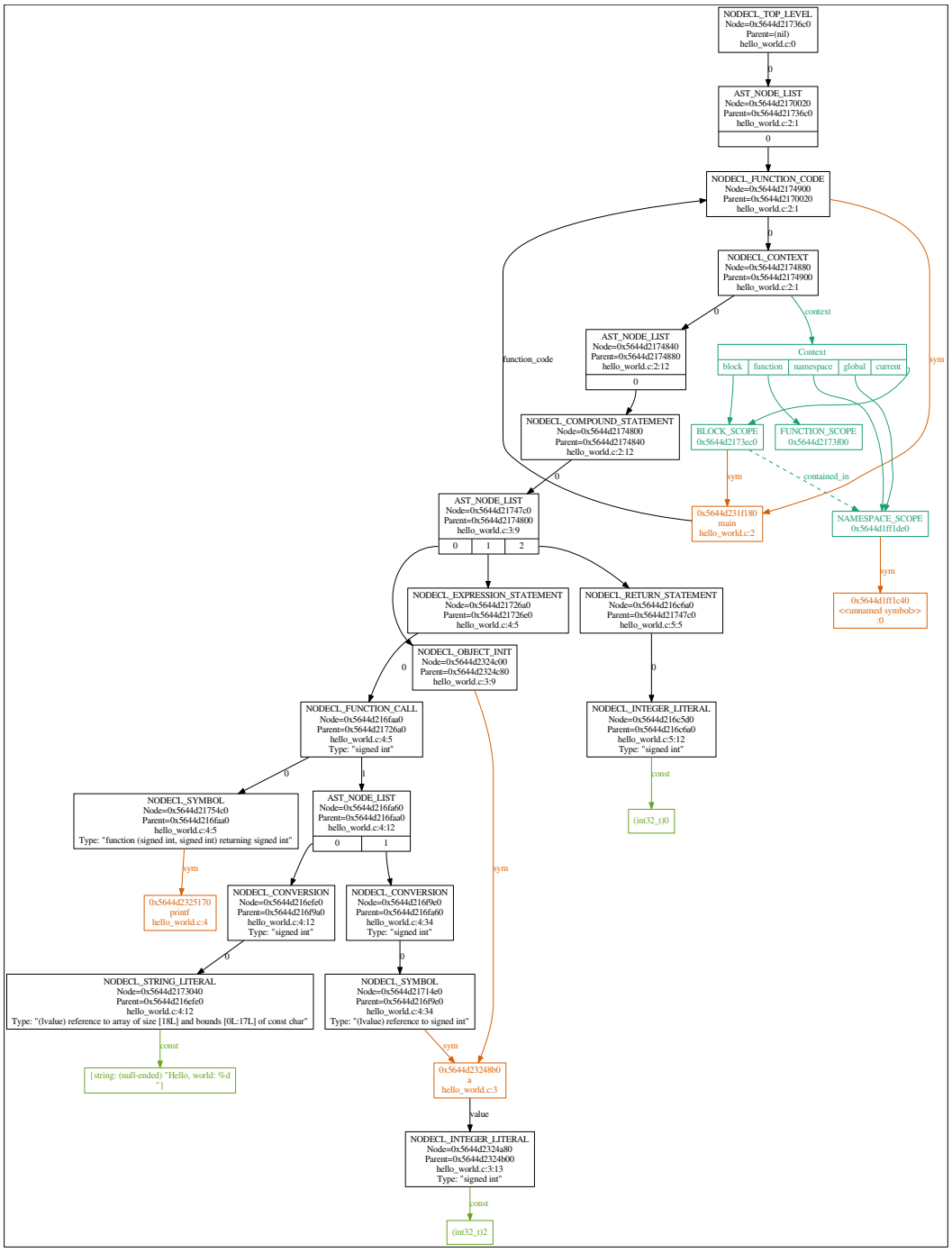

Figure 6.1: Resultant AST for the example

to empower the phase and make it able to perform code transformations with the translation units it has access to. In order to do so, we need to create another class –similarly to any phase– that inherits from a class called `Nodecl::ExhaustiveVisitor`.

The code of the `Nodecl::ExhaustiveVisitor` class is generated with an in-house-developed Python tool, when *autoconf* is used to generate the required Makefile and source code, and contains all the instances of `virtual` functions needed to traverse each possible node, called `visit-pre()`, `visit()` and `visit-post()` this functions can be replaced by the visitor's homonymous functions to implement the functionality required by the phase.

The visit functions are just being called when traversing the tree of `Nodecl` objects that represent the code. When visiting a Node the functions are executed in this order:

1. `visit-pre()` function.
2. `visit()` function.
3. `visit-post()` function.

This can be used in order to perform some special work with the visitor. Nevertheless, while useful, the need to use this functions has never arisen.

Chapter 7

xsmll lowering phase

This chapter describes how the *xsmll* lowering phase is implemented, and how it works internally, using the explanation of the previous chapter (6).

7.1 `LoweringPhase` class

7.1.1 Class infrastructure

Besides the function `run()` where all the logic will have to be implemented, we need to also describe the scaffolding that we had to write in order to support the Mercurium phase interface.

Constructor An imposed limitation to the interface. We need to also set a phase name, and a description. Additionally, we also have defined a parameter that will be used by the phase –the name of the function for which the transformations will be performed, given that we can only apply the transformations to one function–.

Destructor Since we aren't required to do anything from the destructor of the class, we just print a notification for the programmer.

7.1.2 `run()` function

Besides this, there's not much else apart from the `run()` function in the phase, where all the work is performed. What the `run()` function does is:

- Print a debug message if the verbosity flag is active.
- Instantiate a *Lower* object which will traverse the tree representing the current translation unit.

The *Lower* class is the one that performs most of the work, and is described thoroughly at its own section (7.2 –`Lower` class–), with the aid of a few ex-

tra classes needed to actually generate the code. These classes are described succinctly here.

ExpandedTaskDependencyGraph This class is part of the analysis phase for the Mercurium compiler as part of the *psocrates* project. It is used to analyze the dependences between tasks in the code. Development and bug-fixes of this project have stemmed from the development of this project, as we depend heavily on this code in order to generate correct code, as it performs the dependence analysis for us.

TaskProperties This class was originally taken from the `Nanos6` phase and modified afterwards. And is used in the phase to analyze the task and having an actual model that represents the task environment (i.e. datasharing, dependences, and other clauses that may be used).

Lower This class is a descendant of the *ExhaustiveVisitor* class, and therefore, it performs the tree traversal in order to generate code and detect certain conditions (explained later).

XsmllCodegen This class is used to keep track of the complete set of tasks of the function on which we are performing the transformation and manages the dependences between these tasks in order to generate correct *xsmll* code. It manages all the other classes and uses them in order to generate the right code.

XthreadCode This class represents the *xsmll* code for a single task, and generates all the code that must be generated, with the aid of the `XsmllCodegen` class. The analysis of the task environment is handled by the `TaskProperties` class, of which it contains one instance.

SharedMemAllocator Since we need to keep track of all the shared memory, we wrote this class in order to encapsulate it out of the `XsmllCodegen` class, and it is used in order to represent the whole of the shared memory in the compiler.

SharedMemVar This class represents a single variable and the space it takes in the shared memory of the *xsmll* runtime.

7.2 `Lower` class

Being a descendant of `ExhaustiveVisitor`, this class implements almost all of the visiting work. It contains:

- Pointer to `LoweringPhase` (the Mercurium phase for *xsmll* code generation).
- One instance of `XsmllCodegen` (class that handles code generation for all tasks).

- Instance of and `AnalysisBase` and `ExpandedTaskDependencyGraph`, which are required in order to perform task analysis to uncover dependences between tasks.

Besides elements in the class, we have also defined four Nodecl visitors. This visitors are described here, and they always check whether or not the node it is visiting is inside the function defined by the user.

7.2.1 `TaskCall` visitor

The code that the `TaskCall` node represents is just a call to a function which is an outline task. This node is currently not supported for transformations. We leave the support for this node as future work. Correct code could be generated by outlining the function inside an xthread and using its frame in order to get the function parameters.

For now, the `TaskCall` node is just substituted by an actual call to its function, in order to keep the correct functionality of the code. Additionally, the compiler issues a warning to let the programmer know that this function will not be invoked as a task.

7.2.2 `Taskwait` visitor

Since we can't also support barriers of the OmpSs model, and we cannot keep the `Nodecl` textual representation in the node, it is currently deleted. This behavior we assume that will not change in the future, as per the *data-flow* paradigm, there's not really a need to use explicit barriers (given proper analysis for code generation, and that the code that has to be transformed complains to the constraints defined).

7.2.3 `FunctionCode` visitor

This visitor is one of the most important of the phase of the phase, as it is used as an entry point for the phase work.

If the name of the function that is represented by the node that we are visiting is the same that the one from the parameter, then we proceed with the transformation procedure, which is:

- Initiate the process in order to have the task dependency graph, and set the pointer to the resulting object in the `XsmllCodegen` object.
- Instantiate a auxiliary class called `TaskRegisterer`, this class is another `ExhaustiveVisitor`, and has only one visitor for the `Task` nodes. This class adds the task to the `XsmllCodegen` object so that it can take into account all the tasks before generating the code (this is required to generate correct code). What this process does is described in section 7.3.1 (`XsmllCodegen` class).

- It also then calls the `walk()` method so that the visitor keeps visiting the statements inside the function, so that the `Lower` visitor con keep visiting its other nodes.

7.2.4 OpenMP `Task` visitor

The last `Nodecl` visitor implemented for the `Lower` class is the one for the `Task` node, and the most important, as it is used to generate the *xsmll* code. Is does the following:

- The visitor first calls the a member function of the `XsmllCodegen` instance, which generates the whole set of code required to support the *xsmll* runtime (this is explained in section 7.3 –`XsmllCodegen` and `XthreadCode` classes–). This generated code is used in the next steps of the visitor.
- It prepends the xthread struct and function at global scope, as they have already been generated as described in section 5.2.
- It prepends the `xschedule()` call in the scope of the task, as it has been described at section 5.3.1.
- The next step, is to append, if this is the last task in the scope, all the `xdecrease()` calls required by the task's scope. This is also described at 5.3.2
- At last, the current task node is substituted by all the code needed in order to pass variables to the xthread. This code is generated in the `XthreadCode` and its generation will be explained in section 7.3.

With that, we conclude describing the `Nodecl` visitors that our phase has, and we proceed to explain the other four classes that are used in the context of this project.

7.3 `XsmllCodegen` and `XthreadCode` classes

This pair of classes are the ones used for code generation. The functionalities of this classes have been divided. The `XthreadCode` class is used to generate the *xsmll* code required for a specific task, while the `XsmllCodegen` works at a higher level, handling all the different `XthreadCode` instances, and adding and removing dependences for the tasks as is fit to generate correct *xsmll* code.

7.3.1 `XsmllCodegen` class

This class is a singleton class, and it is instantiated only once (when the `Lower` class is instantiated).

It contains the following data:

LoweringPhase* phase This is a pointer to the `LoweringPhase` that has instantiated this class.

ExpandedTaskDependencyGraph* ETDG This is pointer to the single instance of the `ExpandedTaskDependencyGraph` used by the class –which is generated from the `FunctionCode` visitor.

SharedMemAllocator mem_allocator It contains a single instance of a `SharedMemAllocator` instance which is used to keep track of all the variables that are shared (represented by a `SharedMemVar` instance), and its position and size at the *xsmll* shared memory.

xthread_info This is a map which uses smart pointers to securely encapsulate all the `XthreadCode` instances, and relate it with the task that it represent. It is needed for instance, have access to the symbol of the xthread descriptor used to the task.

scope_info This is another map used to relate a `Task` node with its current scope. It is used to keep track of its position in the scope. For each scope, we can append the `xdecrease()` function call after the `Task` node if the task is the last one in the scope.

Once instantiated, this class is not used until the `FunctionCode` visitor of the Lower object calls the function `XsmllCodegen::register_task()` for every task in the function that needs to be transformed (this is done with the aid of the `TaskRegisterer` class, as described in section 7.2.3.

Registering tasks

This function basically does the following, for every task in the function:

- Perform the instantiation and initialization of the `XthreadCode` that represents the current task, and add it to the task map –`XsmllCodegen` has a `std::map` used to relate the `Task` nodes with a `XthreadCode` instance–. When instantiating of the `XthreadCode` object, it creates the xthread descriptor symbol, because it may be needed when generating code for any other task.
- Populate a `std::map` that relates a given `TL::Scope` with the set of tasks contained in the scope it represents.
- Populate the `SharedMemAllocator` instance with the shared variables of the task. This is done with the help of the `TaskProperties` instance of the `XthreadCode` object that represents this task, and is used to keep track of the size and position of the different shared variables used in the function, to be able to generate correct code which will call the `xsubscribe ()` function using its size and offset.

Generating xthread code

After this process has concluded, the next time this object is used from the OpenMP Task visitor (of the `Lower` class), where we need to generate the code for the `Task` node that we're dealing with. This is done with a function called `XsmllCodegen::gen_xthread_code()`. This function uses the `XthreadCode` instance which represents the current task to generate the code, and this functionality is described in section 7.3.2 (`XthreadCode` class).

7.3.2 `XthreadCode` class

This class is the one that actually generates the code, and there is one instance of this class for every task found in the function for which we are performing the transformation. This class is the most complex of the phase. As described in section 7.2.4 (OpenMP `Task` visitor) this generated code is immediately used from the Task visitor. This code mirrors the steps described in the Code generation chapter (5).

Handling inter-task dependences

Given that we need to know and use the dependences between the tasks, all this information must be already handled before starting generating code.

This is achieved by executing a first function that uses the results of the `ExtendedTaskDependencyGraph` to sort out the dependences between the different xthreads.

This task dependency graph has one node for each task instance, if it has to be executed more than once –for example, for loops whose iterations are distributed among parallel task will have multiple instances of the task–. Since in our current limitations we cannot represent different instances of the same task –limitations described at section 4.1 (Usage of *xsmll* shared memory)–, we need to flatten the graph and work for all the nodes for a single task. If any one of them depends on a different task, the code that it will generate will be incorrect.

In this process we evaluate the tasks that depend on the current one –output dependences–, and the tasks on which the current one depends –input dependences–. For each one we do the following:

output dependences We use this dependences in order to know what xthread descriptors we will have to call `xdecrease()` on. To that, we store the symbol of the xthread that we must call `xdecrease()` on and add it as a `firstprivate` dependency. That way, the field will be added to the struct that represents the frame.

input dependences We use the number to generate the initial value of the *synchronization counter* of the xthread that represents this task (if the

xthread has no input dependences, then we simply use 1 as the initial synchronization counter value, and add the `xdecrease()` call at the end of the scope of the task. Otherwise, we just use the number of dependences as the initial value of the synchronization counter, and we'll append the `xdecrease()` call when needed at the end of another xthread –this will happen because this task will be an output dependency of the tasks that will have the `xdecrease()` function call.

Once we have handled all the required xthread dependences, we can start generating the actual code. The functions that do all this are called just after handling the inter-task dependences.

Generating xthread frame

The function `XthreadCode::gen_xstruct()` uses all the dependences and datasharings of the variables to generate the global code that will represent the frame struct used in the xthread. At a high level, it basically:

- Generates the object that represents the `struct`, its name will just be "xsmll_struct_${tasknumber}_s". Where ${tasknumber} represents the actual numerical value of the task as it is added to the map with the `XsmllCodegen::register_task()` function (this notation will be used from now on).
- For every firsprivate variable, it adds the field to the struct.
- Similarly, for every shared variable, it adds the field to the struct, but it also changes the type of the variable to be that of a pointer to the specified type.
- Finally, we generate a `Nodecl` instance that represents the struct definition, and store it in the `XsmllCodegen` instance.

Generating xthread function

The next thing to be created is the function that represents the xthread. This function, as seen in section 5.2.2, is composed of three different parts, the function preface, its body, and the final epilogue. The steps to generate the function are:

Preface

- Generate a symbol for the function, its name will be, similarly to the `struct`, "xsmll_task_${tasknumber}".
- Create a statement of initialization of a variable which will get the frame of the xthread through the `xpreload()` function call. This variable's name will be "f_${tasknumber}_data".

- Iterate through the frame, and for each field, generate a symbol with the same name as the field and take its value.
- If needed, generate declaration of the `firstprivate` variables to take space in the function stack.

Body

- Perform a deep-copy of the body of the task.
- With another `ExhaustiveVisitor` class, we can substitute the usage of variable which are shared automatically because we are now using pointers to the elements, and we can modify the body of the code so that it dereferences pointers as needed.
- Lastly, we use another `ExhaustiveVisitor` subclass, which takes care of removing redundant dereferentiation or referentiation.

Epilogue

- We generate a call to the `xpublish()` function for shared variables –which must also be out dependences for correct code generation– so that the runtime deals with the shared variables, and append it to the code.
- We append the needed calls for the `xdecrease()` function call so that the succeeding xthreads' synchronization counter is decreased and thus the next xthreads may start.
- We finally add the `xdestroy()` call so that the *xsmll* runtime can free the resources associated with the xthread.

Once all this steps have been performed, the function corresponding to the xthread is already generated, and it is appended to the global scope by the OpenMP Task visitor, as stated multiple times already.

Generating xschedulez function call

The next step when generating the code is the generation of the `xschedulez()` call. This function is created in a very straightforward way, since –from the compiler's point of view– it just needs to get the function definition from global scope, and use the symbol of the xthread –which is generated at the instantiation of the object-, and the number of input dependences if there's at least one of such dependent tasks –1 if there are no input dependences–.

Generating xinstantiation

When generating the xthread-related code, we have defined as "xinstantiation" the set of code that is needed in order to copy variables to the frame, as well as reserving its shared memory. In order to generate this code:
We have seen at section 5.4.2 (Shared variables) how we have to generate code when dealing with shared variables. Remember also that we don't have access

to macros, and therefore, we must use a frame at runtime in order to calculate what the position of the field will be. Therefore, if we have any `firstprivate` or `shared` variable, we'll have to add a call to `xpoststor()`.

- First we'll have to generate a call to `xpoststor()` if we have to generate any code to pass variables to the frame.
- If we have any shared variable, then we use the `SharedMemAllocator` instance at the `XsmllCodegen` class to create the `xsubscribe()` call for this variable.
- The next step is to append all this calls to `xsubscribe()` to the xinstantiation.
- Finally, we add the code which passes the variables using the `firsprivate` datasharing. We use the symbol that represents the frame, and generate assignments to all the fields which use are `firsprivate`. This also includes the xthread descriptors that are output dependences. This step is done with the help of the `TaskProperties` class.
  - One curious bug happened at this point: Since the task dependency analysis code performs code analysis for loops, a task iteration may depend on another different iteration. So we could assign the very same xthread descriptor than that of the current xthread. This error was fixed as soon as it was found out.

Generating `xdecrease()` function call

With most of the code being generated, we only require to generate the `xdecrease()` function call. Since, the `xdecrease()` function call for this xthread is already generated if this task depends on another tasks(s). –as seen in section 7.3.2 (Generating xthread function)–. Therefore, the `xdecrease()` function call will only be generated if the current task doesn't depend on any other task –if it is a root task–, and it will decrease the synchronization counter by a value of 1.

7.4 `SharedMemAllocator` and `SharedMemVar` classes

These two last classes were written to take into account the size of the different objects that are using the runtime shared memory.

Basically we keep one instance of `SharedMemAllocator` which will contain a `std::vector` of `SharedMemVar` instances, and a map for symbols and `SharedMemVar` instances.

The `SharedMemAllocator` keeps track of the size that has been used so far. In order to calculate how much space a variable takes, we make use of the

`DataReference` class of the Mercurium compiler internal API. When adding a symbol (which is represented by a `SharedMemVar` instance), we set the current total size as the offset of the added variable, and update the total taken so far with the size of the added variable.

When required, we can call the `SharedMemVar::create_xsubscribe_call()` function, and it will generate the call to the function call that will take care of subscribing the region of memory using the correct size and offset.

Chapter 8

Results

After describing the structure of the compiler phase in chapter 7, we proceed to describe the actual results for this project.

8.1 Testing of the Mercurium phase

We have written a Mercurium phase that deals with transformations from code annotated with OmpSs to code ready to run with *xsmll*.

There are a pair of shortcomings with the *xsmll* phase in its current stage. The first one is related with the semantics of the `xdecrease()` function. We found a bug with the *xsmll* runtime that was promptly fixed, but we came to know that the code generated that made use of the `xdecrease()` was wrong. This is related to the way this function should be used, as it should be used whenever a frame is modified, as it makes the runtime flush the partial results for the frames, making the runtime update all the instances of the frame for the *xsmll* nodes. We should then generate code that calls `xdecrease()` whenever there's a change in any frame.

The other shortcoming is related to the usage of the dependences. Given a loop which has tasks inside, we currently do not support correct transformation for this tasks. This is left as future work.

As part of the verification and developmental phases of the stage, we wrote a battery of tests to check that the transformations are performed correctly, as well as *xsmll* programs using the runtime in a less "dataflowy" fashion (such as the "barrier thread" described in section 4.1 –Lack of barrier primitives–).

8.2 Benchmarks

We also wrote a set of problems to check that the whole transformation were correctly working. These problems are *array summation*, *dot product*, and *matrix multiplication*. We opted to execute the problem with two sizes, a small one, and a big one (which could not be bigger due to memory constraints at compile time caused by the analysis phase). The reason for having two sizes is that the runtime has a relatively high overhead, and we also wanted to see if it scaled well for reasonable problem sizes.

Since we couldn't generate correct *xsmll* code with tasks in loops (as described at section 8.2 (Benchmarks), we opted to write all the applications consisting of 4 different replications of the same problem. Each replication of the problem deals with its own data and is comprised of 3 tasks, one that initializes the data, another one that performs the computation, and finally, another one that prints the results of such problem.

We used four replications of the problems hoping that its execution would work for 4 different nodes. Unfortunately, we did not get it running successfully. We suspect that the runtime may have some race conditions that impede the correct execution in the runtime. Luckily, we managed to get an approximate number of cycles (the results given to us by the cotson simulator are not as accurate as they could be) for one and two nodes, and we can calculate the actual speedup of using one nodes versus two nodes. And with the execution results we see that all the problems are executed without problem.

The number of cycles of each execution is shown at table 8.1, these results are also represented graphically at figure 8.1. We can see that it just has a real positive speed up for the big matmul case. When the problem is small, we could even get negative slow down in the number of cycles executed. This shows us that there's a relatively high overhead for the runtime.

Problem	Size	Nodes	Cycles ($x10^6$)	Speedup
`array-sum`	Small	1	2334	
		2	2331	0.99871
	Big	1	2454	
		2	2403	1.02122
`dot-product`	Small	1	2313	
		2	2331	0.99227
	Big	1	2448	
		2	2415	1.01366
`matmul`	Small	1	2364	
		2	2430	1.02791
	Big	1	9513	
		2	5961	1.59587

Table 8.1: Runtime results of the tests with the *xsmll* runtime

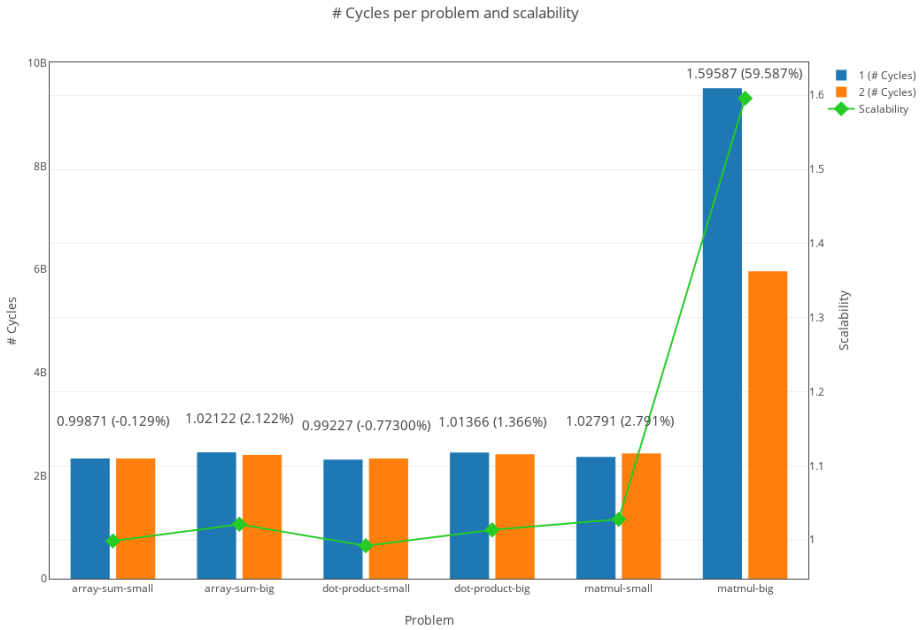

Figure 8.1: Graphical view of the results

Chapter 9

Conclusions and future work

In this Master Thesis, we have written a phase for the Mercurium compiler that can perform simple source-to-source transformations from code using OmpSs annotations to code using the *xsmll* runtime. In order to do so, we spent some time understanding the examples given to us, writing test applications with the *xsmll* runtime as well as testing out different ideas for implementing tasking with *xsmll*.

While writing the Mercurium compiler phase we found defects in the *xsmll* runtime, as well as the Mercurium compiler analysis phase. These defects were reported to the corresponding developers, and helped them with the fixing, discussing ideas about its design and usage .

In the end we have written a phase that can correctly generate code for the *xsmll* runtime, for OmpSs tasks, with certain constrains.

In order to test the runtime we used three problems (sum of arrays, dot product, and matrix multiplication). These problems where tested with light and heavy computation loads, and we proved that the runtime with heavier computation scales better, since there appears to be a relatively high overhead when running with light loads.

As a future work –it is currently almost finished– we have to generate correctly the code for tasks after a loop, as it currently doesn't work properly. We also have to take into account the semantics of `xdecrease()` and change the code generation accordingly. Additionally, it would be interesting to get execution traces of the *xsmll* applications, for further analysis. In order to do that, in the future, we plan to port the Extrae runtime tool to the *xsmll* runtime.

List of Figures

List of Tables

List of Listings

Bibliography

[1] Eduardo Argollo et al. "COTSon: Infrastructure for Full System Simulation". In: *SIGOPS Oper. Syst. Rev.* 43.1 (Jan. 2009), pp. 52–61. ISSN: 0163-5980. DOI: 10.1145/1496909.1496921. URL: http://doi.acm.org/10.1145/1496909.1496921.

[2] Programming Models group at BSC. *OmpSs Examples and Exercises*. URL: https://web.archive.org/web/20170908091723/https://pm.bsc.es/ompss-docs/examples/index.html (visited on 09/08/2017).

[3] Programming Models group at BSC. *OmpSs Specification*. URL: https://web.archive.org/web/20170912072310/https://pm.bsc.es/ompss-docs/spec/OmpSsSpecification.pdf (visited on 09/12/2017).

[4] Partnership for Advanced Computing in Europe. *PATC: Heterogeneous Programming on GPUs with MPI & OmpSs*. URL: https://web.archive.org/web/20170908091642/https://www.bsc.es/patc-2018-mpi-and-ompss (visited on 09/08/2017).

[5] Roberto Giorgi et al. "TERAFLUX: Harnessing dataflow in next generation teradevices". In: *Microprocessors and Microsystems* 38.8 (2014), pp. 976 – 990. ISSN: 0141-9331. DOI: http://dx.doi.org/10.1016/j.micpro.2014.04.001. URL: http://www.sciencedirect.com/science/article/pii/S0141933114000490.

[6] Roberto Giorgi Paolo Gai. *D5.2 – Remote Memory Access (available upon request)*. Feb. 1, 2017.

[7] Xavier Martorell Roberto Giorgi. *D4.1 – Programming Model Extensions (available upon request)*. Feb. 9, 2016.

[8] Xavier Martorell Roberto Giorgi. *D4.2 – AXIOM Code Generation and Instrumentation (available upon request)*. Feb. 7, 2017.

[9] Sara Royuela et al. "Compiler analysis for OpenMP tasks correctness". In: *Proceedings of the 12th ACM International Conference on Computing Frontiers, CF'15, Ischia, Italy, May 18-21, 2015*. 2015, 7:1–7:8. DOI: 10.1145/2742854.2742882. URL: http://doi.acm.org/10.1145/2742854.2742882.